by
Ray Duncan

Power Programming

Implementing Encapsulation and Inheritance in C++

■ C++'s support for subclassing is primitive—there's no smart programming environment that understands and maintains the class hierarchy.

As you're probably tired of hearing by now, the three hallmarks of object-oriented programming languages (OOPLs) are encapsulation, inheritance, and polymorphism. In the last column, I introduced you to C++'s support for encapsulation of data and code inside classes and objects using the `struct`, `union{}`, `class{}`, `private`, `protected`, and `public` keywords. Here, we'll review C data types and C++ encapsulation briefly before moving on to discuss C++'s implementation of inheritance.

DATA STRUCTURES IN C

In C, new data types are built out of existing data types with the `struct{}` keyword. For example, you could create a new POINT data type whose member variables specified a point's location with a declaration in the following form:

```
struct POINT {
    int X;
    int Y;
};
```

You'd then *instantiate* the data type—that is, allocate named storage for some points using the `struct{}` declaration as a template—with statements in the form

```
struct POINT pointA, pointB;
```

Such data declarations follow the usual C scoping rules. If a declaration is outside of any procedure, the data is allocated statically in the data segment and can be accessed by all procedures in that module. On the other hand, if a declaration is inside a procedure, the data is allocated on the stack and is "visible" by name only within that procedure (although the data can, of course, be passed as a parameter in calls to other procedures).

The C support for structures has two serious weaknesses. First, the language is not truly extensible: you can't declare totally novel data types; you can only build new data types out of C's "hard-wired" native data types. Second, there is no provision for overloading the language's native operators to handle new `struct{}` data types in a natural manner. Both of these weaknesses have been answered in various ways by features of C++.

ENCAPSULATION IN C++

C++ classes are declared much like structures. In fact, the `struct{}` keyword still exists in C++, but has been made much more powerful, and it has two sister keywords—`union{}` and `class{}`—which have the same syntax but slightly different semantics. A C++ class includes both variables and the functions that operate on those variables in the same declaration. For example, we could define a C++ class POINT, which would include both a point's location and a "member function" to set the location:

```
// definition of class POINT
struct POINT {
    int X;
    int Y;
    void setXY( ... ) ...
};
```

Actually, the member function can be coded either in-line, like this:

```
// definition of class POINT
struct POINT {
    int X;
    int Y;
    // member function to set
    // point location
    void setXY(int NewX, int NewY)
    {
        X = NewX;
        Y = NewY;
    }
};
```

Or, it can be coded "out-of-line," in the form

```
// definition of class POINT
struct POINT {
    int X;
    int Y;
    void setXY(int, int);
};

// member function to set point
 location
void POINT::setXY(int NewX, int
 NewY);
{
    X = NewX;
    Y = NewY;
};
```

The selection of in-line or out-of-line declaration affects the compiler's compilation of the code for the function as in-line code (like a macro expansion) or a called procedure. Notice the C++ scoping

operator—the double-colon (::)—in the out-of-line form. The POINT modifier and the scoping operator in the definition of setXY() remind the compiler that the visibility of setXY() is determined by the class POINT to which it belongs, and that setXY()'s access to member variables and member functions is the same as the access enjoyed by other members of the POINT class.

Either way, once the POINT class is defined, we can instantiate a point object—allocate named storage for a point—and set the point's location with code like this:

```
POINT pointA;
pointA.setXY(5,10);
```

In this case, the dot (.) operator is a member function pointer; it tells the compiler to invoke the function named setXY() in the class to which the object pointA belongs. Thus, dot is somewhat analogous to the C language's -> operator for addressing member variables in C structures.

Let's return to the encapsulation angle for a moment. The member variables and functions of a C++ class can each have one of three possible attributes: public, protected, or private. A *public* variable or function is accessible by functions elsewhere in the program according to ordinary C-like scoping constraints. *Protected* means that the variable or function is visible only to other members of the same class, or to members of derived (child) classes (we'll return to this shortly when we get to the subject of inheritance). A *private* variable or function is completely invisible outside the class.

We're now in a better position to understand how the class declaration keywords `struct{}`, `union{}`, and `class{}` differ. They have identical syntax, but slightly different effects with regard to encapsulation:

- The member variables and functions of a `struct{}` are public by default, but can be made protected or private as needed.
- The members of a `union{}` are all public, all the time.
- The members of a `class{}` are private by default but can be made public or protected as needed.

Our POINT class example seems to work well enough at first glance, but it's severely out of line with the OOP encapsulation dogma. In true object-oriented programming, objects are supposed to be black boxes, and their data is never visible externally. Objects can only operate on each other's data indirectly, by sending "messages" to each other's member functions. In order to become theologically correct, we can use the `class{}` and `public` keywords to revise the definition of POINT as follows:

The means of making class relationships and the visibility of functions explicit in C++ are bound up in a cryptic syntax that rockets C's reputation for unreadability into a whole new dimension.

```
// definition of class POINT
class POINT {
    int X;
    int Y;
  public:
    void setXY(int, int);
};

// member function to set point
 location
void POINT::setXY(int NewX, int
 NewY)
{
    X = NewX;
    Y = NewY;
};
```

Any instantiations of this version of the POINT class are true objects in the classic sense. Each object has its own copies of the member variables X and Y (hence the term *instance variables*). The member variables are `private` (because that is the `class{}` default), so they can only be accessed by member functions of the same class. In contrast, the member function setXY() has been given the **`public`** attribute, so it can be called from anywhere in the program, limited only by the usual visibility rules. As a result, an external procedure has only one way to modify the values of X and Y in a POINT object: invoking (sending a message to) the POINT class's setXY() function.

INHERITANCE IN C++

We're now well positioned to talk about C++ inheritance. I'll keep the examples simple, but don't feel bad if you find the notation obscure—C++'s support for subclassing is as primitive as all of its other object-oriented features. In Smalltalk, encapsulation and inheritance are built into the programming environment as well as into the programming language: subclassing is supported by code browsers and inspectors that display the relationships between classes graphically and directly and facilitate the cloning, extension, and modification of class functions and variables in a very natural way. In C++, there is no programming environment that is "smart" about the programming language, and the burden of understanding and maintaining the class hierarchy is shifted entirely to the programmer. The relationships among classes and the visibility of class functions and variables must be made explicit on every occasion, and the means of doing so are bound up in a cryptic syntax that rockets C's reputation for unreadability into a whole new dimension.

A new class is derived from an existing class in C++ with a construct in the following general form:

```
class ParentClass {
    ...
};

class ChildClass : [access]
 ParentClass {
    ...
};
```

As you can see, when the child class is defined, its name is followed by a colon (:) separator and then by an optional *access modifier* and the name of the parent or *base* class. The access modifier determines the visibility of inherited variables and functions to users or descendants of the child class—in other words, what access rights will be passed on by the child class—it does not have any effect on the attri

POINT.H — **COMPLETE LISTING**

```
// POINT.H -- Class definition for points and pixels.
// Member functions are defined in POINT.CPP
// Copyright (C) 1991 Ziff Davis Communications
// PC Magazine * Ray Duncan

// definition of base class POINT
class POINT {
  private:
    double X;                          // X coordinate for point
    double Y;                          // Y coordinate for point
    double deg2rad(double);            // convert degrees to radians
    double rad2deg(double);            // convert radians to degrees
  public:
    POINT(double, double);             // constructor for point
    void setX(double);                 // set point value of X
    void setY(double);                 // set point value of Y
    double getX(void);                 // return point value of X
    double getY(void);                 // return point value of Y
    void translate(double, double);    // translate point by X,Y
    void rotate(double);               // rotate point by X,Y
};

// definition of derived class PIXEL
class PIXEL : public POINT {
    int Color;                         // pixel color value
  public:
    PIXEL(double, double, int);        // constructor for pixel
    void setColor(int);                // set color of pixel
    int getColor(void);                // return color of pixel
    void display(void);                // display data for pixel
};

const double pi = 3.141592654;         // constant Pi
```

Figure 1: This header file shows a class definition for two-dimensional points.

utes that were assigned in the parent class. If the access modifier is `private`, any variables and functions inherited from the parent become invisible to users of the child class, regardless of their original attribute. If the access modifier is `public`, the visibility of inherited variables and functions is determined by their original attribute as declared in the parent class. The default access modifier for child classes defined with `class{}` is `private`, and the default for those defined with `struct{}` is `public`.

Let's look at a simple example of inheritance. Imagine that we have all the code for our POINT class safely debugged, compiled, and stowed away in a library, and then find ourselves involved in a graphics project where we need a class of PIXEL objects, each with a location and a color. If we were still programming in C, we'd have to resort to frighteningly low-level mechanisms of code reuse: We'd copy the files containing the POINT source code to new files with different names, edit the code until it approximated our new requirements, and then enter the additional compile-link-debug cycle. Of course, the mere fact that we decided to turn an editor loose on previously proven source code means that we'd have all sorts of opportunities to introduce syntax or spelling errors that are completely tangential to the problem at hand, let alone the usual logical errors of omission or commission.

In C++, on the other hand, we can reuse our existing POINT code in a much more elegant manner. We create a PIXEL class by subclassing the POINT class directly, allowing the new class to inherit as much code and data from the old class as possible, and writing additional code only for the capabilities that differentiate the new class from the old. For example, to create a PIXEL class that uses the location data of the POINT class and adds a member variable for color along with a function to set its value, we could derive the PIXEL class as follows:

```
// definition of base class POINT
class POINT {
    int X;
    int Y;
  public:
    void setXY(int, int);
};

// member function to set point
 location
void POINT::setXY(int NewX, int
 NewY)
{
    X = NewX;
    Y = NewY;
};

// definition of derived class
 PIXEL
class PIXEL : public POINT {
    int Color;
  public:
    void setColor(int);
};

// member function to set pixel
 color
void PIXEL::setColor(int NewColor)
{
   Color = NewColor;
};
```

We can instantiate some pixels:

```
PIXEL pixelA, pixelB;
```

and then set their locations and colors:

```
pixelA.setXY(20,100);
pixelA.setColor(1);
```

From the point of view of the user of the PIXEL class, the fact that setXY() was actually defined in an ancestor class (along with the variables X and Y that it operates on) is completely transparent. The user of the PIXEL class concerns himself only with the capabilities exported by the PIXEL class and not whether those capabilities are inherent in the class or inherited from another class; he really doesn't need to know where the actual work is occurring or the actual location where the pixel is being stored.

There are many possible variations on this theme, depending on how you decide to assign access rights in the parent and child classes. For instance, if we wanted to create a member function in the PIXEL class that would set a pixel's location and color in a single operation, we could write

```
// definition of derived class
 PIXEL
class PIXEL : public POINT {
    int Color;
  public:
    void setPixel(int, int, int);
};
```

POINT.CPP — **COMPLETE LISTING**

```
// POINT.CPP
// Definitions of Member Functions for Point and Pixel Classes
// Copyright (C) 1991 Ziff Davis Communications
// PC Magazine * Ray Duncan

#include <math.h>
#include <iostream.h>
#include "point.h"

// Member function for POINT class to set X coordinate
void POINT::setX(double NewX)
{
    X = NewX;
};

// Member function for POINT class to set Y coordinate
void POINT::setY(double NewY)
{
    Y = NewY;
};

// Member function for POINT class to fetch X coordinate
double POINT::getX(void)
{
    return(X);
};

// Member function for POINT class to fetch Y coordinate
double POINT::getY(void)
{
    return(Y);
};

// Constructor function for POINT class
POINT::POINT (double NewX, double NewY)
{
    X = NewX;
    Y = NewY;
};

// member function for POINT class to translate point by given X,Y
void POINT::translate(double DispX, double DispY)
{
    X = X + DispX;
    Y = Y + DispY;
}

// member function for POINT class to rotate point by given degrees
void POINT::rotate(double degrees)
{
    double radians = deg2rad(degrees);
    double tempX = (X * cos(radians)) - (Y * sin(radians));

    Y = (X * sin(radians)) + (Y * cos(radians));
    X = tempX;
}

// private member function for POINT class to convert degrees to radians
double POINT::deg2rad(double degrees)
{
    return ((degrees * 2 * pi)/360);
}

// private member function for POINT class to convert radians to degrees
double POINT::rad2deg(double radians)
{
    return ((radians * 360)/(2 * pi));
}

// Member function for PIXEL class to set color
void PIXEL::setColor(int NewColor)
{
    Color = NewColor;
};

// Member function for PIXEL class to fetch color
int PIXEL::getColor(void)
{
    return(Color);
};

// Member function for PIXEL class to display location and color
void PIXEL::display(void)
{
    cout << "  X = " << getX();
    cout << "  Y = " << getY();
    cout << "  Color = " << getColor();
};

// Constructor function for PIXEL class
PIXEL::PIXEL (double NewX, double NewY, int NewColor) : POINT (NewX, NewY)
{
    Color = NewColor;
};
```

Figure 2: This listing contains the member functions for the two-dimensional POINT class. The class supports getting and setting the point location, displaying the point location, and translating or rotating points.

```
// member function to set pixel
 location and color
void PIXEL::setPixel(int NewX, int
 NewY, int NewColor)
{
    setXY(NewX, NewY);
    Color = NewColor;
};
```

This is probably the "cleanest" approach, but the nested function calls slow it down. For better performance, we could rewrite the POINT class definition with the `protected` keyword to make its member variables available to descendant classes, then have setPixel() assign new values to X and Y directly:

```
// definition of base class POINT
class POINT {
  protected:
    int X;
    int Y;
  public:
    void setXY(int, int);
};
  ...
```

```
// definition of derived class
 PIXEL
class PIXEL : public POINT {
    int Color;
  public:
    void setPixel(int, int, int);
};

// member function to set pixel
 location and color
void PIXEL::setPixel(int NewX, int
 NewY, int NewColor)
{
    X = NewX;
    Y = NewY;
    Color = NewColor;
};
```

CONSTRUCTORS AND DESTRUCTORS

Constructors and destructors are basic to object-oriented programming languages and confer great power when properly used. Though the terms sound rather arcane and are reminiscent of Saturday morning cartoons, they're actually pretty simple to understand. Basically, every class has special constructor and destructor member functions. The constructor function is called each time the class is instantiated to make a new object, and the destructor function is called whenever an object is destroyed (for example, when a function that instantiated the object in "automatic" storage on the stack terminates, and its stack frame is destroyed).

We've been able to ignore the existence of constructors and destructors until now because C++ provides default constructors and destructors for every class. The default constructor allocates sufficient storage to hold the object's member variables and initializes those variables to zero, and the default destructor merely deallocates the storage that was previously allocated by the constructor.

There will be many cases, however, where we'll want to write our own constructor and destructor functions. For example, we may want to give member variables nonzero initial values or to carry out other, more-complex operations when an object is created or destroyed. The syntax for writing a C++ constructor functic is quite straightforward: You just decla

TRYPOINT.CPP — **COMPLETE LISTING**

```
// TRYPOINT.CPP - Try Point and Pixel Classes
// Compile with Borland C++ 2.0
// Copyright (C) 1991 Ziff Davis Communications
// PC Magazine * Ray Duncan May 1991

#include <iostream.h>
#include "point.h"

main()
{
    int tempX, tempY, tempColor;            // scratch variables
    double degrees;

    PIXEL myPixel(0,0,0);

    cout << "\nEnter X coordinate:   ";     // prompt for pixel
    cin >> tempX;                           // location and color
    cout <<   "Enter Y coordinate:   ";
    cin >> tempY;
    cout <<   "Enter pixel color:    ";
    cin >> tempColor;

    myPixel.setX(tempX);                    // set pixel value
    myPixel.setY(tempY);
    myPixel.setColor(tempColor);

    cout << "\nEnter X translation:  ";     // prompt for pixel
    cin >> tempX;                           // translation and
    cout <<   "Enter Y translation:  ";     // rotation
    cin >> tempY;
    cout <<   "Enter rotation (deg): ";
    cin >> degrees;

    myPixel.translate(tempX, tempY);        // now apply translation
    myPixel.rotate(degrees);                // and rotation factors

    cout << "\nNew value of pixel:  ";     // display new pixel values
    myPixel.display();
}
```

Figure 3: This demonstration program can be linked with POINT.H and POINT.CPP. It prompts you for a pixel location, pixel color, translation distance in the X and Y axes, and number of degrees of rotation, and then displays the resulting point location. The code in Figures 1 through 3 has been tested with Borland C++, Version 2.0.

a member function whose name is exactly the same as the class's name. Suppose we wanted to change our POINT and PIXEL classes so we could assign a location and color when a PIXEL was created, without the need for an additional call to the setPixel() function. We'd add constructor functions to our class definitions as follows:

```
// definition of class POINT
class POINT {
  protected:
    int X;
    int Y;
  public:
    POINT(int, int)
    void setXY(int, int);
};

// constructor for class POINT
POINT::POINT (int NewX, int NewY)
{
    X = NewX;
    Y = NewY;
};

// definition of class PIXEL
class PIXEL : public POINT {
     int Color;
  public:
    PIXEL(int, int);
    void setPixel(int, int, int);
};

// constructor for class PIXEL
PIXEL::PIXEL (int NewX, int NewY,
 Int NewColor)
   : POINT (NewX, NewY)
{
   Color = NewColor;
};
```

Once the constructors are installed in our class definitions, we can instantiate a PIXEL with an initial location of (X,Y)=(-1,-1) and Color=7 like this:

```
PIXEL pixelB(-1, -1, 7);
```

Notice that the constructor for PIXEL invokes the constructor for the ancestor class POINT before carrying out its own actions. In fact, this happens whether you want it to or not: C++ always calls the constructors of all the predecessor classes before calling the constructor for a derived class; this way all inherited variables are properly allocated and initialized when the derived class's constructor starts running. Writing out the call to the constructor explicitly, however, gives you the chance to pass specific values. Also, observe that constructor functions do not have a return type. This is always the case—it's one of the ways that the compiler identifies the definition as a constructor.

Destructor member functions are defined in much the same way as constructors, except that the name of a destructor is always the same as the name of the class with a preceding tilde character (~), for example:

```
// destructor for class PIXEL
PIXEL::~PIXEL()
{
    ....
};
```

Destructor functions have neither a return type nor any parameters.

It's only fair to mention that C++ also allows you to declare special member functions to copy objects and to perform type conversions on member variables, but we'll leave these subjects for another time.

Figures 1 through 3 contain a more complete version of the POINTS and PIXEL classes in the form of a header file, a module containing definitions of the class's member functions, and a demonstration program. This source code is compatible with Borland C++, Version 2.0.

THE IN-BOX

Please send your questions, comments, and suggestions to me at any of the following e-mail addresses:
PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan ■